Kapitel 3: Datenstrukturen und Funktionen in Python

McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. 2. Auflage. Sebastopol, CA [u. a.]: O'Reilly.

Überarbeitet: armin.baenziger@zhaw.ch, 2. Januar 2020

- Bevor wir Add-On-Bibliotheken für die Datenmanipulation besprechen (insb. NumPy und Pandas), werden Funktionalitäten behandelt, die in der Python-Sprache fest implementiert sind und im Kurs regelmässig verwendet werden.
- Wir beginnen mit Pythons zentralen Datenstrukturen: Tupel, Listen, Dicts und Sets.
- Danach werden wir eigene Funktionen schreiben, um den Funktionsumfang von Python zu erweiteren.
- *Den Abschnitt "3.3 Files and the Operating System" werden wir nicht besprechen, da wir im Kurs lediglich Pandas-Funktionen für das Lesen und Schreiben von Datendateien verwenden!*

- Einiges, was in diesem Kapitel thematisiert wird, haben wir bereits im 2. Kapitel kurz besprochen.
- Das Kapitel ist somit eine (hoffentlich gute) Mischung aus Repetition und Erweiterung des Stoffs!

In [1]:
```
%autosave 0
```

```
Autosave disabled
```

Datenstrukturen und Sequenzen

Pythons Datenstrukturen sind einfach, aber leistungsstark. Ihre Verwendung zu meistern ist wichtig.

Tupel (tuple)

Tupel sind **nicht modifizierbare** (*immutable*) Sequenzen von Python-Objekten mit *fester Länge*.

In [2]:
```
tup = (4, 5, 6)
tup
```

Out[2]:
```
(4, 5, 6)
```

In [3]:
```
# es geht auch ohne Klammern:
tup = 4, 5, 6
tup
```

Out[3]:
```
(4, 5, 6)
```

In [4]:
```
# Ein Tupel kann weitere Tupel (oder andere Sequenzen) enthalten:
nested_tup = (4, 5, (5.1, 5.2), 6)
nested_tup
```

Out[4]:
```
(4, 5, (5.1, 5.2), 6)
```

Mit der Funktion `tuple` können Sequenzen/Iteratoren in Tupel konvertiert werden.

In [5]:
```
liste = [4, 0, 2]
tuple(liste)
```

Out[5]:
```
(4, 0, 2)
```

In [6]:
```
tuple(range(5))
```

Out[6]:
```
(0, 1, 2, 3, 4)
```

In [7]:
```
tup = tuple('abcd')
tup
```

Out[7]:
```
('a', 'b', 'c', 'd')
```

Auf Elemente kann mit eckigen Klammern `[]` zugegriffen werden. Wie in vielen anderen Programmiersprachen (z. B. C, C++, Java) hat das erste Element in einer Sequenz den Index 0. (Beispielsweise in *Matlab* oder *R* hat das erste Element hingegen den Index 1.)

In [8]:
```
tup[0]
```

Out[8]:
```
'a'
```

Tupel sind nicht modifizierbar.

In [9]:
```
tup[0] = 'A'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-035c1eb91b05> in <module>()
----> 1 tup[0] = 'A'

TypeError: 'tuple' object does not support item assignment
```

Tupel können mit `+` verknüpft (engl. concatenate) werden.

In [10]:
```
(1, 2, 'drei') + ('vier', 'fünf')
```

Out[10]:
```
(1, 2, 'drei', 'vier', 'fünf')
```

Multiplikation eines Tupel mit einer ganzen Zahl führt - wie bei Listen oder Zeichenketten auch - dazu, dass das Tupel vervielfacht wird.

In [11]:
```
(1, 2) * 3
```

Out[11]:
```
(1, 2, 1, 2, 1, 2)
```

Tupel entpacken (unpacking tuples)

In [12]:
```
tup = (4, 5, 6)
a, b, c = tup  # a ist nun gleich 4, b=5, c=6
a * b * c
```

Out[12]:
```
120
```

Diese Funktionalität erlaubt es auch, Variablennamen sehr einfach zu tauschen. In anderen Sprachen würde man typischerweise wie folgt vorgehen:

In [13]:
```
a, b = 1, 2
# a und b sollen nun getauscht werden:
tmp = a
a = b
b = tmp
print('a ist nun', a, 'und b ist nun', b)
```

```
a ist nun 2 und b ist nun 1
```

In Python kann man das einfacher coden:

In [14]:
```
a, b = b, a  # wieder zurück getauscht
print('a ist nun', a, 'und b ist nun', b)
```

```
a ist nun 1 und b ist nun 2
```

Häufig wird das Entpacken in Iterationen über Sequenzen von Tupeln oder Listen gebraucht.

In [15]:
```
seq = [(6, 7), (1, 5), (3, 1)] # Liste mit drei Tupel
for a, b in seq:
    print(b-a)
```

```
1
4
-2
```

Tupel-Methoden

Da Tupel nicht modifizierbar sind, gibt es nur wenige Methoden auf Tupel. Sehr nützlich ist aber `count` (auch auf Listen anwendbar), welche zählt, wie oft ein bestimmter Wert in einem Tupel vorkommt.

In [16]:
```
a = (1, 2, 2, 3, 2)
a.count(2)
```

Out[16]:
```
3
```

Kontrollfragen:

In [17]:
```
# Gegeben:
tup1 = ('Altdorf', 'Baden', 'Winterthur')
tup2 = ('a', ('b', 'c'), 'd')
print('tup1:', tup1)
print('tup2:', tup2)
```

```
tup1: ('Altdorf', 'Baden', 'Winterthur')
tup2: ('a', ('b', 'c'), 'd')
```

In [18]:
```
# Frage 1: Was ist der Output?
tup1[1]
```

Out[18]:
```
'Baden'
```

In [19]:
```
# Frage 2: Was ist der Output?
tup2[1]
```

Out[19]:
```
('b', 'c')
```

In [20]:
```
# Frage 3: Was könnte wohl nun der Output sein?
tup2[1][0]
```

Out[20]:
```
'b'
```

In [21]:
```
# Frage 3: Was könnte wohl nun der Output sein?
tup1[2][0]
```

Out[21]:
```
'W'
```

Listen (list)

Listen sind im Gegensatz zu Tupel **modifizierbare** (*mutable*) Sequenzen von Python-Objekten mit *variabler Länge*.

- Erzeugung einer Liste mit eckiger Klammer:

In [22]:
```
list_1 = [2, 3, 7, 6]
list_1
```

Out[22]:
```
[2, 3, 7, 6]
```

- Erzeugung einer Liste aus einem anderen Objekt mit der Funktion `list`:

In [23]:
```
tupel  = ('eins', 2, 'drei')
list_2 = list(tupel)
list_2
```

Out[23]:
```
['eins', 2, 'drei']
```

In [24]:
```
list_2[1] = 'zwei'  # Listen sind modifizierbar.
list_2
```

Out[24]:
```
['eins', 'zwei', 'drei']
```

In [25]:
```
iterator = range(10)
iterator
# nur ein "Platzhalter" für die ganzen Zahlen von 0 bis (aber ohne) 10
```

Out[25]:
```
range(0, 10)
```

In [26]:
```
list(iterator)   # Nun haben wir eine Liste.
```

Out[26]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Elemente hinzufügen und entfernen

- Element können mit der Methode `append` am Ende der Liste hinzugefügt werden:

In [27]:
```
list_2.append('vier')
list_2
```

Out[27]:
```
['eins', 'zwei', 'drei', 'vier']
```

Mit `pop` können wir Elemente aus Listen entfernen. Per Default wird das letzte Element gelöscht.

In [28]:
```
list_2.pop()
list_2
```

Out[28]:
```
['eins', 'zwei', 'drei']
```

Mit `in` kann überprüft werden, ob ein Wert in einer Liste vorkommt.

In [29]:
```
'fünf' in list_2
```

Out[29]:
```
False
```

In [30]:
```
'fünf' not in list_2
```

Out[30]:
```
True
```

Mit `+` können zwei Listen verkettet werden.

In [31]:
```
a = [0, 1]
b = [2, 3, 4]
c = a + b
c
```

Out[31]:
```
[0, 1, 2, 3, 4]
```

Sortieren

Listen können mit der `sort` -Methode "in-place" (ohne ein neues Objekt zu kreieren) sortiert werden.

In [32]:
```
a = [7, 2, 5, 1]
a.sort()
a
```

Out[32]:
```
[1, 2, 5, 7]
```

Man kann `sort` einige Argumente übergeben. Beispiele:

- Absteigende Sortierung:

In [33]:
```
a.sort(reverse=True)
a
```

Out[33]:
```
[7, 5, 2, 1]
```

Slicing

Aus den meisten Sequenz-Typen kann man Ausschnitte durch die "Slice-Notation" auswählen. Die grundsätzliche Form lautet `Sequenz[start:stop]` , wobei das Element an der Stelle `stop` nicht mehr zählt.

In [34]:
```
seq = [7, 1, 0, 6, -1, 3]
seq[1:4]
```

Out[34]:
```
[1, 0, 6]
```

Man kann Slices auch zuweisen:

In [35]:
```
seq[1:2] = [99, 999]  # Ein Wert wird durch zwei Werte ersetzt!
seq       # Man beachte: Die Liste enthält nun ein Element mehr.
```

Out[35]:
```
[7, 99, 999, 0, 6, -1, 3]
```

Dies ist nicht identisch mit folgender Syntax, welche eine *verschachtelte* Liste erstellt:

In [36]:
```
seq[1] = [9999, 99999]
seq
```

Out[36]:
```
[7, [9999, 99999], 999, 0, 6, -1, 3]
```

Man kann `start` oder `stop` weglassen, womit dann entweder der Anfang oder das Ende der Sequenz verwendet werden.

In [37]:
```
seq = list('abcdefgh')
seq
```

Out[37]:
```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

In [38]:
```
seq[:3]
```

Out[38]:
```
['a', 'b', 'c']
```

In [39]:
```
seq[3:]
```

Out[39]:
```
['d', 'e', 'f', 'g', 'h']
```

Negative Indizes slicen die Sequenz vom Ende her.

In [40]:
```
seq[-2:]
```
Out[40]:
```
['g', 'h']
```

In [41]:
```
seq[-3:-1]
```
Out[41]:
```
['f', 'g']
```

Nach einem zweiten Doppelpunkt kann ein `step` eingegeben werden, z. B. um jedes zweite Element zu nehmen.

In [42]:
```
seq[1:6:2]
```
Out[42]:
```
['b', 'd', 'f']
```

In [43]:
```
seq[::2]
```
Out[43]:
```
['a', 'c', 'e', 'g']
```

Mit folgendem Trick kann man die *Reihenfolge* einer Liste oder eines Tupels umdrehen:

In [44]:
```
seq[::-1]
```
Out[44]:
```
['h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']
```

Kontrollfragen:

In [45]:
```
# Gegeben:
liste = list('ABCDE')
liste
```
Out[45]:
```
['A', 'B', 'C', 'D', 'E']
```

In [46]:
```
# Frage 1: Was ist der Output?
liste[-1]
```
Out[46]:
```
'E'
```

In [47]:
```
# Frage 2: Was ist der Output?
liste[1:3]
```
Out[47]:
```
['B', 'C']
```

In [48]:
```
# Frage 3: Wie kann der String 'F' der Liste hinzugefügt werden?
liste.append('F')
liste
```
Out[48]:
```
['A', 'B', 'C', 'D', 'E', 'F']
```

In [49]:
```
# Alternative Lösung:
liste.pop()    # zuerst letzten Wert ('F') wieder löschen.
liste = liste + ['F']  # Alternative Lösung
liste
```
Out[49]:
```
['A', 'B', 'C', 'D', 'E', 'F']
```

In [50]:
```
# Frage 4: Ersetzen Sie in "liste" den ersten Buchstaben durch "a":
liste[0] = 'a'
liste
```
Out[50]:
```
['a', 'B', 'C', 'D', 'E', 'F']
```

In [51]:
```
# Frage 5: Was ist der Output?
liste[:2] = ['x', 'y']
liste
```
Out[51]:
```
['x', 'y', 'C', 'D', 'E', 'F']
```

Zwei Sequenz-Funktionen in Python

sorted

Die `sorted` -Funktion gibt eine neue sortierte Liste aus den Elementen einer beliebigen Sequenz zurück:

In [52]:
```
sorted([7, 1, 2, 6, 0, 3, 2])
```

Out[52]:
```
[0, 1, 2, 2, 3, 6, 7]
```

In [53]:
```
sorted('Thomas Muster')
```

Out[53]:
```
[' ', 'M', 'T', 'a', 'e', 'h', 'm', 'o', 'r', 's', 's', 't', 'u']
```

- Die `sorted` **-Funktion** akzeptiert dieselben Argumente wie die `sort` **-Methode** auf Listen.
- Die `sort` -Methode ist im Gegensatz zur `sorted` -Funktion "in-place" (Objekt wird permanent verändert):

In [54]:
```
a = [2, 1, 5]
print(sorted(a))
print(a)      # a ist unverändert geblieben
```

```
[1, 2, 5]
[2, 1, 5]
```

In [55]:
```
a.sort()
print(a)      # a ist permanent verändert (sortiert)
```

```
[1, 2, 5]
```

Kontrollfragen:

In [56]:
```
# Gegeben:
liste = [5, 6, -4]
liste
```

Out[56]:
```
[5, 6, -4]
```

In [57]:
```
# Frage 1: Was ist der Output?
sorted(liste)
```

Out[57]:
```
[-4, 5, 6]
```

In [58]:
```
# Frage 2: Was ist der Output?
liste
```

Out[58]:
```
[5, 6, -4]
```

In [59]:
```
# Frage 3: Was ist der Output?
liste.sort()
liste
```

Out[59]:
```
[-4, 5, 6]
```

reversed

- `reversed` iteriert über die Elemente einer Sequenz in umgekehrter Reihenfolge.
- `reversed` ist ein *Iterator*, so dass er nicht die umgekehrte Sequenz erzeugt, bis er materialisiert wird (z. B. mit `list` oder einer `for` -Schleife).

In [60]:
```
a = [2, 5, 1, 3]
reversed(a)          # Iterator ist angelegt.
```

Out[60]:
```
<list_reverseiterator at 0x137b69d9b70>
```

In [61]:
```
list(reversed(a))    # Mit tuple() gäbe es ein Tupel.
```

Out[61]:
```
[3, 1, 5, 2]
```

dict

- `dict` ist eine weitere wichtige Datenstruktur in Python.
- Es handelt sich um eine flexible Kollektion von Schlüssel-Werte-Paaren.
- `dict` -Objekte können mit geschweiften Klammern `{}` und Doppelpunkten, welche Schlüssel und Werte trennen, erstellt werden.

In [62]:
```
dict1 = {'a' : 'Anna', 'b' : 'Benno'}
dict1
```

Out[62]:
```
{'a': 'Anna', 'b': 'Benno'}
```

In [63]:
```
dict1['a']  # Wert mit Schlüssel a auslesen.
```

Out[63]:
```
'Anna'
```

In [64]:
```
# Werte (values) dürfen auch Listen (oder andere Sequenzen) sein:
dict1
dict1['c'] = ['Claude', 'Claudia']
dict1
```

Out[64]:
```
{'a': 'Anna', 'b': 'Benno', 'c': ['Claude', 'Claudia']}
```

In [65]:
```
dict1['c']
```

Out[65]:
```
['Claude', 'Claudia']
```

In [66]:
```
dict1['c'][0]
```

Out[66]:
```
'Claude'
```

In [67]:
```
'b' in dict1    # Prüft, ob der Schlüssel b in d1 enthalten ist.
```

Out[67]:
```
True
```

Löschen mit `del` oder `pop` :

In [68]:
```
dict2 = dict1.copy()  # Kopie von dict1 erstellen.
```

In [69]:
```
del dict2['b']
dict2
```

Out[69]:
```
{'a': 'Anna', 'c': ['Claude', 'Claudia']}
```

Schlüssel und Werte können mit den Methoden `keys` und `values` ausgelesen werden.

In [70]:
```
dict1.keys()          # Typ "dict_keys"
```

Out[70]:
```
dict_keys(['a', 'b', 'c'])
```

In [71]:
```
list(dict1.keys())    # Keys als Liste
```

Out[71]:
```
['a', 'b', 'c']
```

```
In [72]: dict1.values()          # Typ "dict_values"
```

```
Out[72]: dict_values(['Anna', 'Benno', ['Claude', 'Claudia']])
```

```
In [73]: list(dict1.values())    # Werte als Liste
```

```
Out[73]: ['Anna', 'Benno', ['Claude', 'Claudia']]
```

Kontrollfragen:

```
In [74]: # Gegeben:
         Loehne = {'K1': 4500, 'K2': 5750, 'K3': 7000}
         Loehne
```

```
Out[74]: {'K1': 4500, 'K2': 5750, 'K3': 7000}
```

```
In [75]: # Frage 1: Was ist der Output?
         Loehne['K2']
```

```
Out[75]: 5750
```

```
In [76]: # Frage 2: Was ist der Output?
         list(Loehne.values())
```

```
Out[76]: [4500, 5750, 7000]
```

set

- Ein `set` (Menge) ist eine ungeordnete Kollektion von *unterschiedlichen* Elementen.
- Zwei Möglichkeiten, wie ein `set` erstellt werden kann:

```
In [77]: # Möglichkeit 1:
         {2, 2, 2, 1, 3, 3}    # Es gibt nur drei unterschiedliche Elemente!
```

```
Out[77]: {1, 2, 3}
```

```
In [78]: # Möglichkeit 2:
         tup = (2, 2, 2, 1, 3, 3)    # Liste ginge auch!
         set(tup)
```

```
Out[78]: {1, 2, 3}
```

Mit `set()` lassen sich beispielsweise alle unterschiedlichen Elemente in einer Sequenz auflisten:

```
In [79]: waschmittel = ['Omo', 'Ariel', 'Omo', 'Dash', 'Ariel']
         set(waschmittel)
```

```
Out[79]: {'Ariel', 'Dash', 'Omo'}
```

Oder es lässt sich die Anzahl unterschiedlicher Elemente ermitteln:

```
In [80]: len(set(waschmittel))
```

```
Out[80]: 3
```

List-Comprehensions

- **List-Comprehension** ist ein geschätztes Feature von Python, mit dem eine neue Liste dadurch erstellt wird, dass man die Elemente einer bestehenden Kollektion durch einen Filter auswählt.
- List-Comprehension erlaubt das mit einem Ausdruck der Form:

```
[expr for val in collection if condition]
```

- Die Filter-Bedingung kann auch weggelassen werden.
- Beispiel 1: Konvertierung der Strings in einer Liste in *Grossbuchstaben*

In [81]:
```
Strings = ['Ein', 'kleiner', 'Versuch', 'in', 'Python', '.']
```

Vorbemerkung: Mit der Methode upper() wird ein String in Grossbuchstaben umgewandelt:

In [82]:
```
Strings[4].upper()
```

Out[82]:
```
'PYTHON'
```

In [83]:
```
# Ganze Liste in Grossbuchstaben umwandeln
# Lösung mit for-Loop:
Resultat = []    # Leere Liste initialisieren

for string in Strings:
    Resultat.append(string.upper())

Resultat
```

Out[83]:
```
['EIN', 'KLEINER', 'VERSUCH', 'IN', 'PYTHON', '.']
```

In [84]:
```
# Lösung mit List-Comprehension:
[string.upper() for string in Strings]
```

Out[84]:
```
['EIN', 'KLEINER', 'VERSUCH', 'IN', 'PYTHON', '.']
```

- Beispiel 2: Auswahl von Strings mit Mindestlänge 3 und gleichzeitige Konvertierung in Grossbuchstaben

In [85]:
```
[string.upper() for string in Strings if len(string) >= 3]
```

Out[85]:
```
['EIN', 'KLEINER', 'VERSUCH', 'PYTHON']
```

Kontrollfragen:

In [86]:
```
# Gegeben:
Orte = ['Basel', 'Winterthur', 'Zug', 'Zürich']
Orte
```

Out[86]:
```
['Basel', 'Winterthur', 'Zug', 'Zürich']
```

In [87]:
```
# Frage 1: Was ist der Output?
[len(ort) for ort in Orte]
```

Out[87]:
```
[5, 10, 3, 6]
```

In [88]:
```
# Frage 2: Was ist der Output?
[len(ort) for ort in Orte if ort[0]=='Z']
```

Out[88]:
```
[3, 6]
```

In [89]:
```
# Frage 3: Erstellen Sie aus "Orte" eine Liste, welche je die ersten
# Buchstaben der Ortschaften enthält. Verwenden Sie hierzu eine List-
# Comprehension.
[ort[0] for ort in Orte]
```

Out[89]:
```
['B', 'W', 'Z', 'Z']
```

Es gibt auch Set- und Dict-Comprehensions, auf die wir aber nicht näher eingehen.

Funktionen

Funktionen sind die primäre und wichtigste Methode, um Code in Python zu organsieren und wiederzuverwenden.

- Falls gleicher oder ähnlicher Code mehr als einmal gebraucht wird, kann es lohnenswert sein, eine (wiederverwendbare) Funktion zu schreiben.
- Funktionen machen den Code lesbarer, weil dadurch einer Gruppe von Python-Befehlen einen Namen gegeben wird.
- Funktionen werden mit dem Schlüsselwort `def` eingeleitet.

In [90]:
```
def celsius(fahrenheit):
    return (fahrenheit-32)*5/9
```

In [91]:
```
celsius(100)
```

Out[91]:
```
37.77777777777778
```

Eine Funktion kann auch mehrere Argumente haben:

In [92]:
```
def Summe(x, y):
    return x+y

Summe(2,5)
```

Out[92]:
```
7
```

Man kann einer Funktion auch Default-Werte übergeben:

In [93]:
```
def eine_Funktion(x, y, z=1):  # z=1, falls z nicht übergeben wird.
    return (x + y) * z
```

In [94]:
```
eine_Funktion(2, 3, 4)          # Argumentübergabe durch Position
```

Out[94]:
```
20
```

In [95]:
```
eine_Funktion(4, 3, 2)         # Argumentübergabe durch Position
```

Out[95]:
```
14
```

In [96]:
```
eine_Funktion(z=4, y=3, x=2)   # Argumentübergabe durch Namen
# So ist die Reihenfolge irrelevant.
```

Out[96]:
```
20
```

In [97]:
```
eine_Funktion(2, 3)           # Jetzt gilt der Default z=1
```

Out[97]:
```
5
```

Man kann Funktionen auch verschachteln:

In [98]:
```
Summe(Summe(1,2), Summe(3,4))
```

Out[98]:
```
10
```

In [99]:
```
Summe(celsius(100), celsius(90))
```

Out[99]:
```
70.0
```

Hinweis: Schlüsselwort-Argumente (hier `z = 1`) müssen Positions-Argumente (hier `x, y`) *folgen*, also am Ende stehen.

Kontrollfragen:

In [100]:
```
# Frage 1: Schreiben Sie die Funktion fahrenheit(),
# welche Grad Celsius in Grad Fahrenheit umwandelt.

def fahrenheit(celsius):
    return celsius*9/5 + 32
```

In [101]:
```
# Frage 2: Was muss der Ouput sein?
fahrenheit(celsius(30))
```

Out[101]:
```
30.0
```

Mehrere Funktionswerte zurückgeben

In Python können sehr einfach mehrere Werte durch eine Funktion zurückgegeben werden.

In [102]:
```
def nonsense():
    a = 5
    b = 6
    c = '7'
    return a, b, c

x, y, z = nonsense()
x + y + int(z)
```

Out[102]:
```
18
```

Eigentlich wird lediglich ein Tupel durch die Funktion zurückgegeben, welches dann entpackt wird. Hier die Verdeutlichung:

In [103]:
```
nonsense()
```

Out[103]:
```
(5, 6, '7')
```

Es ist auch möglich ein `dict` zurückzugeben:

In [104]:
```
def add_mult(x1, x2):
    return { 'Summe': (x1 + x2), 'Produkt': (x1*x2) }
```

In [105]:
```
add_mult(2, 3)
```

Out[105]:
```
{'Summe': 5, 'Produkt': 6}
```

Funktionen sind Objekte

Da Python-Funktionen Objekte sind, können viele Konstrukte leicht ausgedrückt werden, die in anderen Sprachen schwierig sind:

In [106]:
```
x = [1, 1, 4, 5, 5, 9]
[min(x), max(x)]        # Liste mit Minimum und Maximum von x
```

Out[106]:
```
[1, 9]
```

Dateien und Betriebssystem

Wir werden im Kurs lediglich Pandas-Funktionen für das Lesen und Schreiben von Datendateien verwenden und besprechen diesen Abschnitt im Lehrmittel somit nicht.

Fazit

- Sie sind nun mit einigen wichtigen Grundlagen der Python-Sprache vertraut.
- Im nächsten Kapitel werden wir die Bibliothek *NumPy* kennenlernen.
- Wir werden sehen, dass man in NumPy mathematische Funktionen auf ganze Arrays anwenden kann, ohne Schleifen schreiben zu müssen.